The `tcp_v4_rcv()` function defined in `ipv4/tcp_ipv4.c` is the main entry point for delivery of datagrams from the IP layer. I thought this test for PACKET_HOST had been done before.

```c
1050 int tcp_v4_rcv(struct sk_buff *skb)
1051 {
1052    struct tcphdr *th;
1053    struct sock *sk;
1054    int ret;
1055
1056    if (skb->pkt_type != PACKET_HOST)
1057       goto discard_it;
1058
1059    /* Count it even if it's bad */
1060    TCP_INC_STATS_BH(TCP_MIB_INSEGS);

As usual, the `pskb_may_pull()` function is responsible for ensuring that the TCP header, and in the next call, the header options are in the `kmalloc'ed` portion of the `sk_buff`.

```c
1062    if (!pskb_may_pull(skb, sizeof(struct tcphdr)))
1063       goto discard_it;
1064
1065    th = skb->h.th;
1066```

Ensure that the offset to the data in words exceeds size of tcp header.

```c
1067    if (th->doff < sizeof(struct tcphdr) / 4)
1068       goto bad_packet;
```

Ensure the options are in the `kmalloc'ed` part.

```c
1069    if (!pskb_may_pull(skb, th->doff * 4))
1070       goto discard_it;
1071```
Some other sanity checks such as data offset and packet length are validated later, but if the checksum (which covers the whole packet) is bad the packet is ditched here.

```c
/* An explanation is required here, I think. */
/* Packet length and doff are validated by header prediction, */
/* provided case of th->doff==0 is eliminated. */
/* So, we defer the checks. */

if ((skb->ip_summed != CHECKSUM_UNNECESSARY &&
    tcp_v4_checksum_init(skb))
    goto bad_packet;
```
Header processing

Attributes of the TCP header are collected in the control buffer. Numeric fields must be converted to host byte order.

- The value of seq is the sequence number of the first byte of this segment. th->ack_seq is the sequence number of the next byte the other end expects to receive from us.

- The value of end_seq is the sequence number just past the last byte carried in this segment.

- The unusual addition of the th->syn and th->fin bitfields is required because each consumes one byte of the sequence number space.

- The subtraction of th->doff * 4 is removing the length of the TCP header.

```c
1080   th = skb->h.th;
1081   TCP_SKB_CB(skb)->seq = ntohl(th->seq);
1082   TCP_SKB_CB(skb)->end_seq =
1083       (TCP_SKB_CB(skb)->seq + th->syn + th->fin +
1084           skb->len - th->doff * 4);
1084   TCP_SKB_CB(skb)->ack_seq = ntohl(th->ack_seq);
1085   TCP_SKB_CB(skb)->when    = 0;
1086   TCP_SKB_CB(skb)->flags   = skb->nh.iph->tos;
1087   TCP_SKB_CB(skb)->sacked  = 0;
1088```
Socket lookup

The \_\_inet\_lookup() function used to be called \_\_tcp\_v4\_lookup(). It attempts to find a struct sock that may be associated with this packet. Lookup elements include source and destination IP and port addresses along with any bound input interface.

```
1089   sk = \_\_inet\_lookup(&tcp\_hashinfo,
1090       skb->nh.iph->saddr, th->source,
1091       skb->nh.iph->daddr, ntohs(th->dest),
1092       inet\_iif(skb));
1093   if (!sk)
1094       goto no\_tcp\_socket;
```

TCP state is checked and the packet is passed to two filters.

```
1096   process:
1097   if (sk->sk\_state == TCP\_TIME\_WAIT)
1098       goto do\_time\_wait;
1099   if (!xfrm4\_policy\_check(sk, XFRM\_POLICY\_IN, skb))
1100       goto discard\_and\_release;
1101   nf\_reset(skb);
1102   if (sk\_filter(sk, skb, 0))
1103       goto discard\_and\_release;
1104   skb->dev = NULL;
1105```

4
Locking considerations

Sockets have a lock structure that is used in different ways depending upon whether the caller is a top-down or bottom-up caller. Bottom up callers block under penalty of death and must use the `bh_lock_sock()` functions. These functions obtain a spinlock that protects the updating of `lock.users` but the value of `lock.users` is the real indicator of whether or not the `sock` is available.

```c
1109    bh_lock_sock_nested(sk);
1110    ret = 0;
```

If the socket is not locked, then `lock.users` will be 0 and `tcp_prequeue()` will be called to attempt to queue the packet on `tp->ucopy.prequeue`. If there is no process blocked waiting to consume data on the socket, then that won't work, and the packet must be processed immediately via the call to `tcp_v4_do_rcv()`. If the socket is locked, the packet must be added to the backlog queue.

```c
1111    if (!sock_owned_by_user(sk)) {
1112        -- net DMA stuff --
1120        {
1121            if (!tcp_prequeue(sk, skb))
1122                ret = tcp_v4_do_rcv(sk, skb);
1123        }
1124    } else
1125        sk_add_backlog(sk, skb);
1126    bh_unlock_sock(sk);
1127    sock_put(sk);
1129    return ret;
```

The `owner` variable is actually a Boolean flag. A value of 0 means that no application process owns the socket at present. A value of 1 means that some application process owns the socket.

```c
#define sock_owned_by_user(sk)  ((sk)->sk_lock.owner)
```
Exit from tcp_v4_rcv()

This is the end of tcp_v4_rcv(). For a "regular" data packet on of three outcomes will have occurred:

- Packet is left on the prequeue (socket not locked and receiver sleeping in tcp_recvmssg())
- Packet is left on the backlog queue (socket locked... receiver active in tcp_send/recvmssg)
- Packet is processed (socket not locked and no receiver sleeping in tcp_recvmssg())

The first two outcomes are "lightweight" in terms of processing.

- In the first case the actual processing which occurs in tcp_v4_do_rcv() is performed in the context of the application that will actually receive the packet.

- In the second case, it will occur in the context of an application that owns the socket.

- In the least desirable third case it will be performed immediately in the context of the softirq. Note that in the first two cases ack generation is also deferred.

- In the COP protocol ALL packet processing is carried out as in the third case.
Handling of unusual conditions

The rest of this code handles exceptional conditions.

```c
1132  no_tcp_socket:
1133  if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
1134     goto discard_it;
1135
1136  if (skb->len < (th->doff << 2)
1137      || tcp_checksum_complete(skb)) {
1137    bad_packet:
1138       TCP_INC_STATS_BH(TCP_MIB_INERRS);
1139    } else {
1140       tcp_v4_send_reset(skb);
1141    }
1142
1143  discard_it:
1144  /* Discard frame. */
1145  kfree_skb(skb);
1146  return 0;
1147
1148  discard_and_relse:
1149  sock_put(sk);
1150  goto discard_it;
1151```
Time wait processing

1152 do_time_wait:
1153    if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb)) {
1154       inet_twsk_put((struct inet_timewait_sock *) sk);
1155       goto discard_it;
1156    }
1163    switch (tcp_timewait_state_process(
                   (struct inet_timewait_sock *)sk,
1164                   skb, th)) {
1165    case TCP_TW_SYN: {
1166       struct sock *sk2 = inet_lookup_listener(&tcp_hashinfo,
1167                  skb->nh.iph->daddr,
1168                  ntohs(th->dest),
1169                  inet_iif(skb));
1170       if (sk2) {
1171          inet_twsk_deschedule((struct inet_timewait_sock *)sk,
1172                               &tcp_death_row);
1173          inet_twsk_put((struct inet_timewait_sock *)sk);
1174          sk = sk2;
1175          goto process;
1176       } /* Fall through to ACK */
1177    }
1179 case TCP_TW_ACK:
1180       tcp_v4_timewait_ack(sk, skb);
1181       break;
1182 case TCP_TW_RST:
1183       goto no_tcp_socket;
1184 case TCP_TW_SUCCESS:
1185    }
1186    goto discard_it;
1187 }

The socket locking mechanism

In this section we take a more detailed look at the locking mechanism.

734 /* Used by processes to "lock" a socket state, so that
735  * interrupts and bottom half handlers won't change it
736  * from under us. It essentially blocks any incoming
737  * packets, so that we won't get any new data or any
738  * packets that change the state of the socket.
739  *
740  * While locked, BH processing will add new packets to
741  * the backlog queue. This queue is processed by the
742  * owner of the socket lock right before it is released.
743  *
744  * Since ~2.3.5 it is also exclusive sleep lock serializing
745  * accesses from user process context.
746 */
747 #define sock_owned_by_user(sk)  ((sk)->sk_lock.owner)
748
Application context socket locking

The `lock_sock()` function may be used by top half callers which are allowed to block. If the number of users is already 1, `__lock_sock()` will be called and the process will block.

In `__lock_sock()` the spinlock will be dropped and retaken after the process wakes up and will return with the lock held.

```c
1534 voidfastcall lock_sock(struct sock *sk)  
1535 {  
1536     might_sleep();  
1537     spin_lock_bh(&sk->sk_lock.slock);  
1538     if (sk->sk_lock.owner)  
1539         __lock_sock(sk);  
1540     sk->sk_lock.owner = (void *)1;  
1541     spin_unlock(&sk->sk_lock.slock);  
1542     /*  
1543      * The sk_lock has mutex_lock() semantics here:  
1544      */  
1545     mutex_acquire(&sk->sk_lock.dep_map, 0, 0, _RET_IP_);  
1546     local_bh_enable();  
1547 }  
```
The `__lock_sock` function.

Use of exclusive wait ensures FIFO (instead of thundering herd) queuing of processes blocked waiting for the socket. Note that it is almost always the case that there will be at most ONE process that is waiting in any case.

```c
1232 static void __lock_sock(struct sock *sk)
1233 {
1234    DEFINE_WAIT(wait);
1235
1236    for(;;) {
1237        prepare_to_wait_exclusive(&sk->sk_lock.wq, &wait,
1238                                     TASK_UNINTERRUPTIBLE);
1239        spin_unlock_bh(&sk->sk_lock.slock);
1240        schedule();
1241        spin_lock_bh(&sk->sk_lock.slock);
1242        if(!sock_owned_by_user(sk))
1243            break;
1244    }
1245    finish_wait(&sk->sk_lock.wq, &wait);
1246 }
```

After wakeup, the lock is obtained again and if the ownership value is 0, then the socket is free. The ownership value will be set to 1 in line 1540 above. This is safe because the spin lock is held ensuring that the the ownership cannot change "out from under us").
The **bh_lock_sock()** facility

The **bh_lock_sock()** macro just obtains the spin lock. The spin lock is held by the caller while the caller checks to see if `lock.users` is 0. If it is not 0, then the bottom half, *which is not permitted to block*, must take a course of action that doesn't require the lock. This action is typically to queue the packet on the *backlog* queue.

In my view the explosive growth of **bh_lock** and **spin_lock** should be controlled. If it can't be controlled, then the entire locking architecture should be rethought.

```c
/* BH context may only use the following locking interface. */
#define bh_lock_sock(__sk)                      
    spin_lock(&(__sk)->sk_lock.slock))

#define bh_lock_sock_nested(__sk) \ 
    spin_lock_nested(&(__sk)->sk_lock.slock), \ 
    SINGLE_DEPTH_NESTING)

#define bh_unlock_sock(__sk)                      
    spin_unlock(&(__sk)->sk_lock.slock))

void __lockfunc _spin_lock_nested(spinlock_t *lock, \ 
    int subclass)
{
    preempt_disable();
    spin_acquire(&lock->dep_map, subclass, 0, _RET_IP_); \ 
    _raw_spin_lock(lock);
}
```
Nested lock handlers -- new to kernel 2.6

This appears to be a new mechanism designed to support holding more than one lock at a time.. possibly a max of two! This is normally done in sane operating systems by using a hierarchical locking scheme.

```c
2366 void lock_acquire(struct lockdep_map *lock,
2367      unsigned int subclass,
2368      int trylock, int read, int check, unsigned long ip)
2369 {
2369      unsigned long flags;
2370      if (unlikely(current->lockdep_recursion))
2371         return;
2372      raw_local_irq_save(flags);
2373      check_flags(flags);
2374      current->lockdep_recursion = 1;
2375      __lock_acquire(lock, subclass, trylock, read, check,
2376      _irqs_disabled_flags(flags), ip);
2377      current->lockdep_recursion = 0;
2378      raw_local_irq_restore(flags);
2379    }
```

```c
299 #ifdef CONFIG_DEBUG_LOCK_ALLOC
300  #ifdef CONFIG_PROVE_LOCKING
301    #define spin_acquire(l, s, t, i)
302      lock_acquire(l, s, t, 0, 2, i)
303  #else
304    #define spin_acquire(l, s, t, i)
305      lock_acquire(l, s, t, 0, 1, i)
306  endif
307  #endif
308  #define spin_release(l, n, i)
309     lock_release(l, n, i)
310#endif
```

The socket release mechanism.

When the socket lock is released, if the backlog queue is not empty, then __release_sock() is called to drain the backlog queue and pass the packets to tcp_v4_do_rcv() for processing. If there are additional processes sleeping on the lock, since this is an EXCLUSIVE wait, one is awakened after the lock is released.

Note that an additional wait queue, sk->sk_lock.wq is associated with the socket.

```c
1551 void fastcall release_sock(struct sock *sk)  
1552 {
1553    /*
1554     * The sk_lock has mutex_unlock() semantics:
1555     */
1556    mutex_release(&sk->sk_lock.dep_map, 1, _RET_IP_);
1557    spin_lock_bh(&sk->sk_lock.slock);
1558    if (sk->sk_backlog.tail)
1559        __release_sock(sk);
1560    sk->sk_lock.owner = NULL;
1561    if (waitqueue_active(&sk->sk_lock.wq))
1562        wake_up(&sk->sk_lock.wq);
1563    spin_unlock_bh(&sk->sk_lock.slock);
1564 } 
```
Draining the backlog queue

Arriving packets are placed on the backlog queue when the sock is locked at arrival time. When the sock is unlocked they are passed to the tcp_v4_do_rcv(). To preserve packet order and improve efficiency, this function must be called before the lock.owner field is reset to 0 via the call to bh_unlock_sock().

Note that it uses queue stealing to assume ownership of the backlog queue before releasing exclusive ownership of the socket. This function is invoked only in top_half (application) context. The lock is dropped while the stolen queue is being drained and additional packets may be added to the backlog queue. The outer loop allows for multiple queue steals.

```c
static void __release_sock(struct sock *sk)
{
    struct sk_buff *skb = sk->sk_backlog.head;

    do {
        sk->sk_backlog.head = sk->sk_backlog.tail = NULL;
        bh_unlock_sock(sk);

        do {
            struct sk_buff *next = skb->next;
            skb->next = NULL;
            sk->sk_backlog_rcv(skb);

            /* We are in process context here with softirqs disabled, use cond_resched_softirq() to preempt.
             * This is safe to do because we've taken the backlog queue private:
             */
            cond_resched_softirq();
            skb = next;
        } while (skb != NULL);
    } while((skb = sk->sk_backlog.head) != NULL);

    bh_lock_sock(skb);
}
```

Here sk->backlog_rcv() is tcp_v4_do_rcv().
The TCP prequeue mechanism.

The prequeue is managed via the `ucopy` substructure of the `tcp_sock`.

```c
319         /* Data for direct copy to user */
320         struct {
321             struct sk_buff_head     prequeue;
322             struct task_struct   *task;
323             struct iovec         *iov;
324             int               memory;
325             int               len;
326         } ucopy;
```

*prequeue:* The queue itself  
*task:* Task struct of process sleeping on receive queue  
*iov:* Iovec associated with pending receive  
*memory:* Total amount of memory occupied by prequeue packets  
*len:* Remaining space in user supplied buffer.
Prequeue processing

When a packet arrives, the first objective is to simply place it on the &tp->ucopy.prequeue. The sk->lock must be held on entry to this function. If tp->ucopy.task is NULL, then there is no sleeping task to wake up and the prequeue is bypassed. Hence the only time the prequeue can be used is when there is a process blocked waiting to receive data on this socket.

The use of the prequeue permits data to be processed in the context of the application to which it will eventually be delivered.

```c
840 static inline int tcp_prequeue(struct sock *sk, 
        struct sk_buff *skb) 
841 { 
842    struct tcp_sock *tp = tcp_sk(sk); 
843 
844    if (!sysctl_tcp_low_latency && tp->ucopy.task) { 
845        __skb_queue_tail(&tp->ucopy.prequeue, skb); 
846        tp->ucopy.memory += skb->truesize; 
847 
848    if (tp->ucopy.memory > sk->sk_rcvbuf) { 
849        struct sk_buff *skb1; 
850 
851        BUG_ON(sock_owned_by_user(sk)); 
852        while ((skb1 = __skb_dequeue(&tp->ucopy.prequeue)) 
853            != NULL) {
854            sk->sk_backlog_rcv(skb, skb1); 
855            NET_INC_STATS_BH(LINUX_MIB_TCPREQUEUEDROPPED); 
856        }
857        tp->ucopy.memory = 0; 
```
Unblocking the application

If the amount of data on the prequeue doesn't exceed the buffer quota, and this is the first packet added to the prequeue, the application is awakened. Presumably, the objective of all of this is to cause as much processing as possible to be done in the context of the application that is actually receiving the data.

```c
858     } else if (skb_queue_len(&tp->ucopy.prequeue) == 1) {
859         wake_up_interruptible(sk->sk_sleep);
860         if (!inet_csk_ack_scheduled(sk))
861             inet_csk_reset_xmit_timer(sk, ICSK_TIME_DACK,
862                                         (3 * TCP_RTO_MIN) / 4,
863                                         TCP_RTO_MAX);
864     }
865     return 1;
866 }
867     return 0;
868 }
```
The `sk_add_backlog()` function

This function adds a packet to the tail of the backlog queue. *The sk->sk_lock.spinlock must be held prior to calling this function!*

```c
471 static inline void sk_add_backlog(struct sock *sk,
    struct sk_buff *skb)
472 {
473    if (!sk->sk_backlog.tail) {
474        sk->sk_backlog.head = sk->sk_backlog.tail = skb;
475    } else {
476        sk->sk_backlog.tail->next = skb;
477        sk->sk_backlog.tail = skb;
478    }
479    skb->next = NULL;
480 }
```
The `tcp_recvmsg()` function.

This function is the analog of `udp_recvmsg()`. It is handles the `recvmsg()` system call and thus executes in an application context.

This routine copies from a sock struct into the user buffer.

Technical note: in 2.3 we work on _locked_ socket, so that tricks with *seq access order and skb->users are not required. Probably, code can be easily improved even more.

```c
1095 int tcp_recvmsg(struct kiocb *iocb, struct sock *sk,
    struct msghdr *msg,
    size_t len, int nonblock, int flags, int *addr_len)
1096 {
    struct tcp_sock *tp = tcp_sk(sk);
    int copied = 0;
    u32 peek_seq;
    u32 *seq; /* Pointer to value holding sequence number of head of unread data */
    unsigned long used;
    int err;
    int target; /* Read at least this many bytes */
    long timeo;
    struct task_struct *user_recv = NULL;
    int copied_early = 0;

    The caller will block here until the `struct sock` can be locked.

    lock_sock(sk);
    TCP_CHECK_TIMER(sk);
```

20
State verification

It is not permissible to receive data on a socket that is in the LISTEN state.

```c
1113    err = -ENOTCONN;
1114    if (sk->sk_state == TCP_LISTEN)
1115       goto out;
1116
1117    timeo = sock_rcvtimeo(sk, nonblock);
1118
1119    /* Urgent data needs to be handled specially. */
1120    if (flags & MSG_OOB)
1121       goto recv_urg;
```

Setting the u32 *seq pointer

The value of *seq is the next sequence number to be consumed by the application code. A comment in the code calls it the ``head of yet unread data''. It normally points to permanent holder in the tcpsock, but for peek requests it is redirected to a local variable.

```c
1123    seq = &tp->copied_seq;
1124    if (flags & MSG_PEEK) {
1125       peek_seq = tp->copied_seq;
1126       seq = &peek_seq;
1127    }
1128
1129    target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
1130```

The objective of the low water test is to set the target variable which is the minimum number of bytes that must be consumed before this tcp_recvmsg completes.

```c
1129    target = sock_rcvlowat(sk, flags & MSG_WAITALL, len);
1130```

Back in sock_initdata() the value of sk->sk_rcvlowat was set to 1 word.

```c
1250 static inline int sock_rcvlowat(const struct sock *sk,
1251                                  int waitall, int len)
1252 { return (waitall ? len : min_t(int,
1253                               sk->sk_rcvlowat, len)) ? : 1;
1253 }```
The main receive loop.

This is the main receive loop. This loop contains subsections that process the different receive queues, and plenty of goto's that make life more interesting.

```c
    do {
       struct sk_buff *skb;
       u32 offset;

       * Are we at urgent data? Stop if we have read anything or have SIGURG pending. */
       if (tp->urg_data && tp->urg_seq == *seq) {
          if (copied) break;
          if (signal_pending(current)) {
             copied = timeo ? sock_intr_errno(timeo) : -EAGAIN;
             break;
          }
       }
    }
```
The receive queue.

Packets are placed in the *receive queue* by *tcp_v4_do_rcv*. The first step is to try to consume a buffer from the *receive* queue. Unlike the backlog queue, prequeue and out of order queue. Packets in the *receive queue* are

(1) already acked  
(2) guaranteed in order  
(3) contain no holes but  
(4) apparently may contain overlapping data

If the *receive* queue is not empty *rcv_nxt* will be the next byte beyond its end.
Processing the receive queue

If the receive queue is empty then the prequeue and the backlog queue will be processed. The “before” test is possible because segments are forced to be in order on the receive_queue. If the test fails means the first byte of the packet on the receive queue has a sequence number that is after the next byte of unconsumed data ... i.e. there is a hole.

```c
1156       /* Next get a buffer. */
1157
1158       skb = skb_peek(&sk->sk_receive_queue);
1159       do {
1160          if (!skb)
1161             break;
1162          /* Now that we have two receive queues this
1163           * shouldn't happen. */
1166          if (before(*seq, TCP_SKB_CB(skb)->seq)) {
1167             printk(KERN_INFO "recvmsg bug: copied %X 
1168                    seq %X\n", *seq, TCP_SKB_CB(skb)->seq);
1169             break;
1170          }
1171          offset = *seq - TCP_SKB_CB(skb)->seq;
1172          if (skb->h.th->syn)
1173             offset--;  
1174          if (offset < skb->len)
1175             goto found_ok_skb;
1176          if (skb->h.th->fin)
1177             goto found_fin_ok;
1178          BUG_TRAP(flags & MSG_PEEK);
1179          skb = skb->next;
1180       } while (skb != (struct sk_buff *)
1181            &sk->sk_receive_queue);
```

Presumably the normal case here is for offset to be 0, but it might be a positive number if this segment contains both retransmitted and new data. If the segment contains all retransmitted data it shouldn't have been placed on this queue in the first place. At any rate the value of offset indicates where in this sk_buff the new data not yet consumed starts.

In the normal case, the goto will cause the subsequent queue processing loops to be passed over.
Testing for “receive complete” conditions

Falling into this section means that either the receive queue was empty or it didn’t contain any usable data. Although the comment indicates that it is necessary to process the backlog queue, there is no evidence of the backlog queue actually being processed here.

What is actually happening is various conditions are tested to see if this receive operation should be allowed to return to the user. The value of copied is initially 0. It seems to normally represent the number of bytes that have been copied to user space, but also as a catchall to indicate error situations. The value of target is the minimum number of bytes that must (normally) be returned to the user.

If at least target bytes have been consumed and the backlog queue is empty, then an exit from the main loop is made.

```
1182     /* Well, if we have backlog, try to process it now yet. */
1183
1184     if (copied >= target && !sk->sk_backlog.tail)
1185        break;
1186
1187     if (copied) {
1188        if (sk->sk_err ||
1189           sk->sk_state == TCP_CLOSE ||
1190           (sk->sk_shutdown & RCV_SHUTDOWN) ||
1191           !timeo ||
1192           signal_pending(current) ||
1193           (flags & MSG_PEEK))
1194        break;
```
1195 } else { // no data copied
1196     if (sock_flag(sk, SOCK_DONE))
1197         break;
1198
1199     if (sk->sk_err) {
1200         copied = sock_error(sk); // copied overloaded :-(
1201         break;
1202     }
1203
1204     if (sk->sk_shutdown & RCV_SHUTDOWN)
1205         break;
1206
1207     if (sk->sk_state == TCP_CLOSE) {
1208         if (!sock_flag(sk, SOCK_DONE)) {
1209            /* This occurs when user tries to read
1210               * from never connected socket.
1211               */
1212            copied = -ENOTCONN;
1213            break;
1214         }
1215         break;
1216     }
1217
1218     if (!timeo) {
1219         copied = -EAGAIN;
1220         break;
1221     }
1222
1223     if (signal_pending(current)) {
1224         copied = sock_intr_errno(timeo);
1225         break;
1226     }
1227 }

The call to tcp_cleanup_rbuf() is primarily concerned with the fact that it may be necessary to schedule a window update type ack if the application has now consumed some data.

1229     tcp_cleanup_rbuf(sk, copied);
Establishing the copy to user space parameters

Arrival at this spot indicates

- the receive queue is empty,
- no serious errors or state changes were noted and
- we haven't consumed sufficient data to return to the caller.

The `ucopy.task` variable appears to be used to ensure that when data is copied to user space it is being copied in the context of the process that first requested the data! Needless to say this would be a good idea. The value of `user_recv` is initially set to NULL at the front end of this function.

Presumably the normal case is for both to be NULL here on the first trip through the loop and to not change thereafter. If two processes are consuming from the same socket and one of them is already sleeping on the receive queue, the `tp->ucopy.task` will not be NULL here. The value of `len` is the user specified buffer length which is saved here in `tp->ucopy.len`.

```c
1230 1231    if (!sysctl_tcp_low_latency && tp->ucopy.task == 
1232          user_recv) {
1233        /* Install new reader */
1234        if (!user_recv && !(flags & (MSG_TRUNC | MSG_PEEK))) {
1235            user_recv = current;
1236            tp->ucopy.task = user_recv;
1237            tp->ucopy.iov = msg->msg_iov;
1238        }
1239    tp->ucopy.len = len;
1240
1241    BUG_TRAP(tp->copied_seq == tp->rcv_nxt || 
1242        (flags & (MSG_PEEK | MSG_TRUNC)));
Testing the prequeue for empty

1243  /* Ugly... If prequeue is not empty, we have to
1244   * process it before releasing socket, otherwise
1245   * order will be broken at second iteration.
1246   * More elegant solution is required!!!
1248   *
1249   * Look: we have the following (pseudo)queues:
1250   *
1251   * 1. packets in flight
1252   * 2. backlog
1253   * 3. prequeue
1254   * 4. receive_queue
1255   *
1256   * Each queue can be processed only if the next ones
1257   * are empty. At this point we have empty receive_queue.
1258   * But prequeue _can_ be not empty after second iteration,
1259   * when we jumped to start of loop because backlog
1260   * processing added something to receive_queue.
1261   * We cannot release_sock(), because backlog contains
1262   * packets arrived _after_ prequeued ones.
1263   *
1264   * Shortly, algorithm is clear --- to process all
1265   * the queues in order. We could make it more directly,
1266   * requeueing packets from backlog to prequeue, if
1267   * is not empty. It is more elegant, but eats cycles,
1268   * unfortunately.
1269   */

This location is where the diagnostic
Apr 21 12:17:27 vmlnx2b kernel: tcp_recvmsg: release prequeue 1
was inserted.

1270          if (!skb_queue_empty(&tp->ucopy.prequeue))
1271                  goto do_prequeue;
1272          /* __ Set realtime policy in scheduler __ */
1273      }
Processing the backlog queue

Arrival here implies that the prequeue was empty. The release/lock combination here causes the contents of backlog queue to be passed to tcp_v4_do_rcv() which will (possibly??) copy them to user space or possibly place them on the receive queue or the out of order queue.

A previous test was:

1184       if (copied >= target && !sk->sk_backlog.tail)
1185          break;

Therefore, arrival here with copied >= target implies the backlog queue is non-empty and we already have enough data to return to the caller. If copied < target we don't know how much data might be on the backlog queue so we just sleep here and depend on tcp_v4_do_rcv() or tcp_prequeue() to wake us up at the appropriate time. This backlog release never occurred in the monitored connection.

1276       if (copied >= target) {
1277          /* Do not sleep, just process backlog. */
1278          release_sock(sk);
1279          lock_sock(sk);
Waiting for data

The caller will sleep here but the sock will be released before sleeping and locked just after wakeup. This used to be called tcp_data_wait()

```c
else
    sk_wait_data(sk, &timeo);

int sk_wait_data(struct sock *sk, long *timeo)
{
    int rc;
    DEFINE_WAIT(wait);
    prepare_to_wait(sk->sk_sleep, &wait, TASK_INTERRUPTIBLE);
    set_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);
    rc = sk_wait_event(sk, timeo,
                       skb_queue_empty(&sk->sk_receive_queue));
    clear_bit(SOCK_ASYNC_WAITDATA, &sk->sk_socket->flags);
    finish_wait(sk->sk_sleep, &wait);
    return rc;
}
```

This is a clever macro that could probably simplify the of "wait" functions in COP.

```c
#define sk_wait_event(__sk, __timeo, __condition) ({
    int rc;
    release_sock(__sk);
    rc = __condition;
    if (!rc){
        *__timeo = schedule_timeout(*(__timeo));
    }
    lock_sock(__sk);
    rc = __condition;
    rc;
})
```
Adjusting the length of the available buffer space.

Arrival here means that we either

- released the backlog queue or
- woke up after waiting for the receive queue to be non-empty

So hopefully there is new data in the receive queue now.

If `user_recv` is not NULL here then it will point to the task struct of the current process. Here `len` is the length of the user supplied buffer. `tp->ucopy.len` was initially set to `len` but it would appear that it may be decremented in `tcp_rcv_established`. The `NET_ADD_STATS` call is designed to accumulate how many bytes get delivered in this way.

```c
1287     if (user_recv) {
1288         int chunk;
1289
1290         /* __ Restore normal policy in scheduler __ */
1291
1292         if ((chunk = len - tp->ucopy.len) != 0) {
1293             NET_ADD_STATS_USER(LINUX_MIB_TCPDIRECTCOPYFROMBACKLOG, chunk);
1294             len -= chunk;
1295             copied += chunk;
1296         }
```
Processing the prequeue

This test ensures that every thing that has been received correctly and in order has also been consumed (i.e. the receive queue is empty). That will not be so if the draining of the backlog queue caused new packets to end up on the receive queue and in that case the prequeue processing must be deferred! Note that processing the prequeue can also lead to a chunk being consumed by the application.

```
1298          if (tp->rcv_nxt == tp->copied_seq &&
1299              !skb_queue_empty(&tp->ucopy.prequeue)) {
1300      do_prequeue:
1301          tcp_prequeue_process(sk);
1302          if ((chunk = len - tp->ucopy.len) != 0) {
1303              NET_ADD_STATS_USER(LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
1304                  len -= chunk;
1305                  copied += chunk;
1306          }
1307      }
1308  }
1309 }
```

Draining the prequeue may have caused new stuff to appear on the receive queue. The continue causes a jump back to the top of the loop to test receive queue.

```
1310      if ((flags & MSG_PEEK) && peek_seq != tp->copied_seq) {
1311          if (net_ratelimit())
1312              printk(KERN_DEBUG "TCP(%s:%d): Application bug, race in MSG_PEEK.\n",
1313                  current->comm, current->pid);
1314          peek_seq = tp->copied_seq;
1315      }
1316      continue;
```
Copying data from the receive queue to application space.

Because of the preceding continue arrival here is only via goto from processing the receive queue.

```c
1318     found_ok_skb:
1319     /* Ok so how much can we use? */
1320     used = skb->len - offset;
1321     if (len < used)
1322         used = len;
1323     /* Do we have urgent data here? */
1324     if (tp->urg_data) {
1325         u32 urg_offset = tp->urg_seq - *seq;
1326         if (urg_offset < used) {
1327             if (!urg_offset) {
1328                 if (!sock_flag(sk, SOCK_URGINLINE)) {
1329                     ++*seq;
1330                     offset++;  
1331                     used--;  
1332                 if (!used)
1333                     goto skip_copy;
1334                     goto skip_copy;
1335                 }
1336             } else
1337                 used = urg_offset;
1338         }
1339     }
```
if (!(flags & MSG_TRUNC)) {

--- NETDMA code

err = skb_copy_datagram_iovec(skb, offset, msg-&gt;msg_iov, used);
if (err) {
  /* Exception. Bailout! */
  if (!copied)
    copied = -EFAULT;
  break;
}

Updating buffer pointers and counters

The value of used is the amount just copied from this skb. The tcp_rcv_space_adjust() function is a fairly opaque algorithm designed to dynamically decide how much buffer space is available.

*seq += used;
copied += used;
len -= used;
tcp_rcv_space_adjust(sk);

Where to copy...

This routine doesn't need to keep track of where to put the data as it is being copied. That is done in memcpy_to_iovec() which dynamically adjusts elements of the iovec to indicate how much of each buffer has been used up. See udprecv.pdf

iov-&gt;iov_len -= copy;
iov-&gt;iov_base += copy;
End of the main loop

1384 skip_copy:
1385       if (tp->urg_data && after(tp->copied_seq, tp->urg_seq)) {
1386           tp->urg_data = 0;
1387           tcp_fast_path_check(sk, tp);
1388       }

If more data exists in this sk_buff then back to the top of the loop.

1389       if (used + offset < skb->len)
1390           continue;
1391
1392       if (skb->h.th->fin)
1393           goto found_fin_ok;
1394       if (!(flags & MSG_PEEK)) {
1395           sk_eat_skb(sk, skb, copied_early);
1396           copied_early = 0;
1397       }
1398       continue;
1399
1400     found_fin_ok:
1401     /* Process the FIN. */
1402     ++*seq;
1403     if (!(flags & MSG_PEEK)) {
1404         sk_eat_skb(sk, skb, copied_early);
1405         copied_early = 0;
1406     }
1407     break;
1408   } while (len > 0);
Exit from the main loop

Finally after falling out of the main loop, the prequeue is drained one more time. Here tp->ucopy.len is set to len if copied > 0 and zero otherwise. Finally the tp->ucopy structure is reinitialized to indicate that there is no application process actively trying to consume data.

```c
1410    if (user_recv) {
1411       if (!skb_queue_empty(&tp->ucopy.prequeue)) {
1412          int chunk;
1413
1414          tp->ucopy.len = copied > 0 ? len : 0;
1415
1416          tcp_prequeue_process(skb);
1417
1418          if (copied > 0 && (chunk = len - tp->ucopy.len) != 0) {
1419 Net_Add_Stats_User(LINUX_MIB_TCPDIRECTCOPYFROMPREQUEUE, chunk);
1420          len -= chunk;
1421          copied += chunk;
1422        }
1423    }
1424
1425    tp->ucopy.task = NULL;
1426    tp->ucopy.len = 0;
1427  }
1428```
Return from tcp_recvmsg()

On return the socket must be released which means that the backlog queue must be drained once more..

    /* According to UNIX98, msg_name/msg_namelen are ignored
    * on connected socket. I was just happy when found this 8)
    */

    /* Clean up data we have read: This will do ACK frames. */
    tcp_cleanup_rbuf(sk, copied);

    TCP_CHECK_TIMER(sk);

    release_sock(sk);
    return copied;

out:
    TCP_CHECK_TIMER(sk);
    release_sock(sk);
    return err;

recv_urg:
    err = tcp_recv_urg(sk, timeo, msg, len, flags, addr_len);
    goto out;
Processing the prequeue

The tcp_prequeue_process() function simply dequeues sk_buffs from the prequeue and passes them to tcp_v4_do_rcv(). This function is called with sk->sk_lock.owner = 1. The usage of local_bh_disable/enable is not clear.

```c
989 static void tcp_prequeue_process(struct sock *sk)  
990 {  
991    struct sk_buff *skb;  
992    struct tcp_sock *tp = tcp_sk(sk);  
993    NET_INC_STATS_USER(LINUX_MIB_TCPPREQUEUED);  
995  
996    /* RX process wants to run with disabled BHs, though it is  
997     * not necessary */  
998    local_bh_disable();  
999    while ((skb = __skb_dequeue(&tp->ucopy.prequeue)) != NULL)  
1000       sk->sk_backlog_rcv(sk, skb);  
1001    local_bh_enable();  
1002  
1003    /* Clear memory counter. */  
1004    tp->ucopy.memory = 0;  
1005 }
```
The *tcp_v4_do_rcv()* function

This function is called directly from *tcp_v4_rcv()* when the *tp->user.task* pointer is found to be NULL in *tcp_prequeue*. It is also serves as the *backlog_rcv* function and is called each time a packet is removed from the backlog queue, when the prequeue overflows or by *tcp_prequeue_process*. Since *tcp_recvmsg()* calls both *release_sock()* and *tcp_prequeue_process()* this function may be called in both top end and bottom end contexts!

```c
991 /* The socket must have it's spinlock held when we get
992  * here.
993  *
994  * We have a potential double-lock case here, so even when
995  * doing backlog processing we use the BH locking scheme.
996  * This is because we cannot sleep with the original
997  * spinlock held.
998 */
999 int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
1000 {
1001   if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
1002     TCP_CHECK_TIMER(sk);
1003     if (tcp_rcv_established(sk, skb, skb->h.th, skb->len))
1004        goto reset;
1005     TCP_CHECK_TIMER(sk);
1006     return 0;
1007   }
```
The \texttt{tcp\_rcv\_established()} function

3847 /*
3848 * TCP receive function for the ESTABLISHED state.
3849 *
3850 * It is split into a fast path and a slow path.
3851 * The fast path is disabled when:
3852 * - A zero window was announced from us - zero window
3853 *   probing is only handled properly in the slow path.
3854 * - Out of order segments arrived.
3855 * - Urgent data is expected.
3856 * - There is no buffer space left
3857 * - Unexpected TCP flags/window values/header lengths are
3858 *   received(detected by checking the TCP header against
3859 *   pred\_flags)
3860 * - Data is sent in both directions. Fast path only
3861 *   supports pure senders
3862 * - Unexpected TCP option.
3863 *
3864 * When these conditions are not satisfied it drops into a
3865 * receive procedure patterned after RFC793 to handle all
3866 * The first three cases are guaranteed by proper pred\_flags
3867 * the rest is checked inline. Fast processing is turned on
3868 * in tcp\_data\_queue when everything is OK.
3869 */
The `tcp_rcv_established` function

A variant of Van J's algorithm is implemented here. His original algorithm was intended only for the downcall path, and the part that involves actual delivery to user space is used only when this function is called by the recipient of the data.

```c
3870 int tcp_rcv_established(struct sock *sk,
             struct sk_buff *skb,
3871          struct tcphdr *th, unsigned len)
3872 {
3873    struct tcp_sock *tp = tcp_sk(sk);
3874
3875    /*
3876     * Header prediction.
3877     * The code loosely follows the one in the famous
3878     * "30 instruction TCP receive" Van Jacobson mail.
3879     *
3880     * Van's trick is to deposit buffers into socket queue
3881     * on a device interrupt, to call tcp_recv function
3882     * on the receive process context and checksum and copy
3883     * the buffer to user space. smart...
3884     *
3885     * Our current scheme is not silly either but we take the
3886     * extra cost of the net_bh soft interrupt processing...
3887     * We do checksum and copy also from device to kernel.
3888     */
3890    tp->rx_opt.saw_tstamp = 0;
3891```
3892    /* pred_flags is 0xS?10 << 16 + snd_wnd
3893    * if header_prediction is to be made
3894    * 'S' will always be tp->tcp_header_len >> 2
3895    * '?' will be 0 for the fast path, otherwise pred_flags
3896    * is 0 to turn it off (when there are holes in receive
3897    * space for instance)
3898    * PSH flag is ignored.
3899    */
3900
3901    if ((tcp_flag_word(th) & TCP_HP_BITS) == tp->pred_flags &&
3902       TCP_SKB_CB(skb)->seq == tp->rcv_nxt) {
3903       int tcp_header_len = tp->tcp_header_len;
3904
Van J's magic is now complicated by the possible presence of the timestamp header option which
didn't exist at the time of his 30 instruction tcp receive.

3905    /* Timestamp header prediction: tcp_header_len
3906        * is automatically equal to th->doff*4 due to pred_flal
3907        * pred_flags match.
3908    */
3909
3910    /* Check timestamp */
3911    if (tcp_header_len == sizeof(struct tcphdr) +
3912       TCPOLEN_TSTAMP_ALIGNED) {
3913       __u32 *ptr = (__u32 *)(th + 1);
3914    /* No? Slow path! */
3915    if (*ptr != ntohl((TCPOPT_NOP << 24) |
3916       (TCPOPT_NOP << 16) |
3917       (TCPOPT_TIMESTAMP << 8) | TCPOLEN_TIMESTAMP))
3918       goto slow_path;
3919
3920    tp->rx_opt.saw_tstamp = 1;
3921    ++ptr;
3922    tp->rx_opt.rcv_tsval = ntohl(*ptr);
3923    ++ptr;
3924    tp->rx_opt.rcv_tsecr = ntohl(*ptr);
PAWs stands for Protection Against Wrapped Sequence Numbers. If the time stamp just went down this could well be a delayed duplicate.

```c
3925     /* If PAWS failed, check it more carefully in slow path */
3926         if ((s32)(tp->rx_opt.rcv_tsval -
3927             tp->rx_opt.ts_recent) < 0)
3928             goto slow_path;
3929
3929     /* DO NOT update ts_recent here, if checksum fails
3930        * and timestamp was corrupted part, it will result
3931        * in a hung connection since we will drop all
3932        * future packets due to the PAWS test.
3933        */
3934 }
3935
Updating recent timestamp

The packet must be a standalone ack or window update if len <= tcp_header_len. The value of rcv_wup is the value of rcv_nxt on the most recent window update that was sent. If the receiving process is a bulk sender, it should receive only ACKs and window updates. The comment on the next page explains that header only packets are checksummed on entry. It seems to me that if this process is a bulk sender then tp->rcv_nxt and tcp->rcv_wup should never change!

```c
3936     if (len <= tcp_header_len) {
3937         /* Bulk data transfer: sender */
3938         if (len == tcp_header_len) {
3939             /* Predicted packet is in window by definition.
3940              * seq == rcv_nxt and rcv_wup <= rcv_nxt.
3941              * Hence, check seq<=rcv_wup reduces to:
3942              */
3943             if (tcp_header_len ==
3944                 (sizeof(struct tcphdr) +
3945                   TCPOLEN_TSTAMP_ALIGNED) &&
3946                 tp->rcv_nxt == tp->rcv_wup)
3947                 tcp_store_ts_recent(tp);
3948         }
3949     }
```
**Standaloneack processing**

The packet must be a standalone ack or window update if \( len \leq tcp\_header\_len \). As with cop when an ACK is received it is necessary to:

- free any sk_buffs that were acked
- see if there is pending data that can now be sent
- free the ack packet itself

```c
            /* We know that such packets are checksummed
             * on entry.
             */
            tcp_ack(sk, skb, 0);
            __kfree_skb(skb);
            tcp_data_snd_check(sk, tp);
            return 0;
        } else { /* Header too small */
            TCP_INC_STATS_BH(TCP_MIB_INERRS);
            goto discard;
        }
```
Validating sequence numbers

The value of `rcv_wup` is the last ack number sent. Because of delayed acks in TCP `rcv_nxt` may be beyond `rcv_wup`. The check is made to allow control information with otherwise invalid sequence numbers to get through.

```c
2784 /* Check segment sequence number for validity.
2785 *
2786 * Segment controls are considered valid, if the segment
2787 * fits to the window after truncation to the window. Acceptability
2788 * of data (and SYN, FIN, of course) is checked separately.
2789 * See tcp_data_queue(), for example.
2790 *
2791 * Also, controls (RST is main one) are accepted using RCV.WUP instead
2792 * of RCV.NXT. Peer still did not advance his SND.UNA when we
2793 * delayed ACK, so that hisSND.UNA<=ourRCV.WUP.
2794 */
2795 static inline int tcp_sequence(struct tcp_sock *tp,
        u32 seq, u32 end_seq)
2796 {
2797     return !before(end_seq, tp->rcv_wup) &&
2798     !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
2799 }
```
Data packet processing

Arrival here means we are still in the fast path and that the packet contained data.

The value of `eaten` will be set to 1 if and only if all of the data in this packet is transferred to user space. The value of `tp->copied_seq` is the sequence number of the next byte of data to be copied to user space. If `tp->copied_seq == tp->rcv_nxt`, then everything up to `rcv_nxt` (which is the start of this packet since we are in the fast path) has already been copied to user space and so its safe to copy this packet. The test `len - tcp_header_len <= tp->ucopy.len` is done to ensure that the entire segment will fit in the user buffer.

Finally the copy can't happen unless the `sk_lock.owner` is set to 1 and the `ucopy.task` is the current process. The value of `sk_lock.owner` is 1 during processing of the `backlog_queue` and `tcp_process_prequeue` and `current` will match `ucopy.task`. But when `tcp_v4_do_receive` is called in bottom half processing `current` would not match `ucopy.task` even if the socket is locked and `sk_lock.owner` is 1. *(Note: this situation can't actually happen -- if the socket is locked the packet goes on the backlog queue.)*

The `tcp_copy_to_iovec()` function will update both `tp->copied_seq` and `tp->ucopy.len`. The copy can occur only if the entire packet will fit in the space left in the user buffer.

```c
} else {
    int eaten = 0;
    int copied_early = 0;
    if (tp->copied_seq == tp->rcv_nxt &&
        len - tcp_header_len <= tp->ucopy.len) {
        --- net DMA ---
        if (tp->ucopy.task == current &&
            sock_owned_by_user(sk) && !copied_early) {
            __set_current_state(TASK_RUNNING);
            if (!tcp_copy_to_iovec(sk, skb,
                        tcp_header_len))
                eaten = 1;
    }
```
Packet fully consumed

Presumably, `tcp_copy_to_iovec()` performs checksumming so it is now safe to update the timestamp. The timestamp is updated only if this segment is the first segment beyond the last ack sent because the timestamp is used to compute the RTT.

```c
if (eaten) {
    /* Predicted packet is in window by definition.
     * seq == rcv_nxt and rcv_wup <= rcv_nxt.
     * Hence, check seq<=rcv_wup reduces to:
     */
    if (tcp_header_len ==
        (sizeof(struct tcphdr) +
         TCPOLEN_TSTAMP_ALIGNED) &&
        tp->rcv_nxt == tp->rcv_wup)
        tcp_store_ts_recent(tp);
    __skb_pull(skb, tcp_header_len);
    tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
    NET_INC_STATS_BH(LINUX_MIB_TCPHPHITSTOUSER);
}
```

Copied early is set by NET_DMA processing. The `tcp_cleanup_rbuf()` function is responsible for determining if a window update must be sent.

```c
if (copied_early)
    tcp_cleanup_rbuf(sk, skb->len);
```

This code was inserted just before the `tcp_copy_to_iovec()` and produced the log messages shown.

```c
if (in_interrupt())
{
    printk("ucopy in interrupt! \n");
    tp->ucopy_in_irq += 1;
}
```
Packet not eaten

If the packet is not fully consumed it must go on the receive queue. It is also checksummed and
time stamp processing is performed in yet another place here.

```c
3997         if (!eaten) {
3998             if (tcp_checksum_complete_user(sk, skb))
3999                goto csum_error;
4000
4001         /* Predicted packet is in window by definition.
4002         * seq == rcv_nxt and rcv_wup <= rcv_nxt.
4003         * Hence, check seq<=rcv_wup reduces to:
4004         */
4005             if (tcp_header_len ==
4006                 (sizeof(struct tcphdr) +
                     TCPOLEN_TSTAMP_ALIGNED) &&
4007                 tp->rcv_nxt == tp->rcv_wup)
4008                 tcp_store_ts_recent(tp);
4009
4010             tcp_rcv_rtt_measure_ts(skb, skb);
4011
```

The usage of `sk_forward_alloc` is not clear, but if the size of the buffer (header and data) exceeds it
then a jump into slow path processing is required.

```c
4012         if ((int)skb->truesize > sk->sk_forward_alloc)
4013             goto step5;
4014
4015         NET_INC_STATS_BH(LINUX_MIB_TCPHPHITS);
```

Here is where the packet is placed on the receive queue and `rcv_next` is updated.

```c
4017         /* Bulk data transfer: receiver */
4018         __skb_pull(skb,tcp_header_len);
4019         __skb_queue_tail(&sk->sk_receive_queue, skb);
4020         sk_stream_set_owner_r(skb, sk);
4021         tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
4022     }
4023
```

48
Ack Generation

The call to *tcp_event_data_recv()* deals with delayed ACK generation. The calls to *tcp_ack()* and *tcp_data_snd_check()* deal with processing the ack carried by this packet and possibly sending data from the send queue. The jump to *no_ack* occurs if there is already pending ack that has been scheduled for transmission.

```c
4024       tcp_event_data_recv(sk, tp, skb); 
4025
4026       if (TCP_SKB_CB(skb)->ack_seq != tp->snd_una) {
4027             /* Well, only one small jumplet in fast path... */
4028             tcp_ack(sk, skb, FLAG_DATA);
4029             tcp_data_snd_check(sk, tp);
4030             if (!inet_csk_ack_scheduled(sk))
4031                 goto no_ack;
4032         }
4033
4034         __tcp_ack_snd_check(sk, 0);
4035
4036         no_ack:
4037         
4038         #ifdef CONFIG_NET_DMA
4039         if (copied_early)
4040         __skb_queue_tail(&sk->sk_async_wait_queue, skb);
4041         
4042     #endif
4043
4044     if (eaten)
4045         __kfree_skb(skb);
4046     
4047     else
4048         sk->sk_data_ready(sk, 0);
4049     return 0;
```
The slow path

This path begins with checksumming. It then determines if the packet should be discarded because PAWS failed. Resets are accepted even if PAWS fails.

```
4049 slow_path:
4050    if (len < (th->doff<<2) ||
        tcp_checksum_complete_user(sk, skb))
4051       goto csum_error;
4052
4053    /*
4054    * RFC1323: H1. Apply PAWS check first.
4055    */
4056    if (tcp_fast_parse_options(skb, th, tp) &&
        tp->rx_opt.saw_tstamp &&
        tcp_paws_discard(sk, skb)) {
4057       if (!th->rst) {
4058          NET_INC_STATS_BH(LINUX_MIB_PAWSESTABREJECTED);
4059          tcp_send_dupack(sk, skb);
4060          goto discard;
4061       }
4062    } /* Resets are accepted even if PAWS failed. */
4064    ts_recent update must be made after we are sure
4065    that the packet is in window.
4066    */
4068  }
```
Sequence number validation

If this segment doesn't contain $tp->rcv_nxt$ its necessary to send a duplicate ack unless it does contain a reset flag. Reset always causes instant death of the connection.

```c
4074    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, 
        TCP_SKB_CB(skb)->end_seq)) {
4075        /* RFC793, page 37: "In all states except SYN-SENT, all reset 
4076         * (RST) segments are validated by checking their SEQ-fields." 
4077         * And page 69: "If an incoming segment is not acceptable, 
4078         * an acknowledgment should be sent in reply (unless the RST 
4079         * bit is set, if so drop the segment and return)".
4080        
4081           if (!th->rst) 
4082              tcp_send_dupack(sk, skb); 
4083           goto discard; 
4084        } 
4085 
4086        if(th->rst) { 
4087           tcp_reset(sk); 
4088           goto discard; 
4089        }
4090 
4091        tcp_replace_ts_recent(tp, TCP_SKB_CB(skb)->seq);
4092```

The packet is rejected in the end is before $rcv_wup$ or the start is after the end of the window.

```c
2797 static inline int tcp_sequence(struct tcp_sock *tp, 
    u32 seq, u32 end_seq)
2798 {
2799    return !before(end_seq, tp->rcv_wup) && 
2800        !after(seq, tp->rcv_nxt + tcp_receive_window(tp));
2801 }```
Because of the way the receive window is computed. \( rcv_{\text{nxt}} + \text{receive\_window} = rcv_{\text{wup}} + rcv_{\text{nxt}} \).

488/* Compute the actual receive window we are currently advertising.  
489 * Rcv\_nxt can be after the window if our peer push more data  
490 * than the offered window.  
491 */
492 static inline u32 tcp_receive_window(const struct tcp_sock *tp)
493 {
494        s32 win = tp->rcv_wup + tp->rcv_wnd - tp->rcv_nxt;
495
496        if (win < 0)
497                win = 0;
498        return (u32) win;
499     }
If the packet carries SYN and the sequence number is before $rcv\_nxt$ the packet is considered a delayed duplicate and nothing happens here.

If the sequence number is beyond $rcv\_nxt$, then the connection is reset. The $before()$ function does deal with sequence number wrap. Since this end of the connection believes it is established, why not just ACK and let the other end reset if need be??

```c
4093    if (th->syn && !before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))
4094        {
4095            TCP_INC_STATS_BH(TCP_MIB_INERRS);
4096            NET_INC_STATS_BH(KERN_MIB_TCPABORTONSYN);
4097            tcp_reset(sk);
4098        } return 1;
```

235 /*
236  * The next routines deal with comparing 32 bit unsigned ints
237  * and worry about wraparround (automatic with unsigned
238  * arithmetic).
239 */
240 static inline int before(__u32 seq1, __u32 seq2)
241 {
242        return (__s32)(seq1-seq2) < 0;
243 }
If the *ack* flag is set in the header then *tcp_ack()* is called to process it.

```c
4099 4100 step5:
4101    if(th->ack)
4102        tcp_ack(sk, skb, FLAG_SLOWPATH);
4103
4104    tcp_rcv_rtt_measure_ts(sk, skb);
4105
4106    /* Process urgent data. */
4107    tcp_urg(sk, skb, th);
4108
4109    /* step 7: process the segment text */
4110    tcp_data_queue(sk, skb);
4111
Check to see if its possible to send a new segment because the one just received *acked* new data or if it is necessary to send an *ack*

```c
4112    tcp_data_snd_check(sk, tp);
4113    tcp_ack_snd_check(sk);
4114    return 0;
```

```c
4116 csum_error:
4117    TCP_INC_STATS_BH(TCP_MIB_INERRS);
4118
4119 discard:
4120    __kfree_skb(skb);
4121    return 0;
4122 }
4123 ```
Queuing the new data

The receive queue consists only of in order and non overlapping segments. If this segment doesn’t fall into that category it must go on the out of order queue. If it fills a hole, then a collection of segments from the out of order queue can move to the receive queue.

```c
3128 static void tcp_data_queue(struct sock *sk,
    struct sk_buff *skb)

3129 {
3130    struct tcphdr *th = skb->h.th;
3131    struct tcp_sock *tp = tcp_sk(sk);
3132    int eaten = -1;

Check for no data in the packet.

3134    if (TCP_SKB_CB(skb)->seq == TCP_SKB_CB(skb)->end_seq)
3135       goto drop;
3136
3137    __skb_pull(skb, th->doff*4);
3138
3139    TCP_ECN_accept_cwr(tp, skb);
3140
3141    if (tp->rx_opt.dsack) {
3142       tp->rx_opt.dsack = 0;
3143       tp->rx_opt.eff_sacks =
3144         min_t(unsigned int, tp->rx_opt.num_sacks,
3145             4 - tp->rx_opt.tstamp_ok);
3146    }
```

Update data pointer to point to actual data and then do ECN and sack processing.

55
Direct copy to user space may occur again here

If the sequence number is \textit{rcv\_nxt} then the packet goes on the receive queue or may be copied to user space. The segment will not be queued if the offered window is presently 0. For copy to user space to succeed the same conditions must prevail as before. The lock must be held and the current task must be the ucopy task.

```c
3147    /* Queue data for delivery to the user.  
3148    * Packets in sequence go to the receive queue.  
3149    * Out of sequence packets to the out\_of\_order\_queue.  
3150    */
3151    if (TCP\_SKB\_CB(skb)->seq == tp->rcv\_nxt) {
3152       if (tcp\_receive\_window(tp) == 0)
3153          goto out\_of\_window;
3154
3155       /* Ok. In sequence. In window. */
3156       if (tp->ucopy\_task == current &&
3157           tp->copied\_seq == tp->rcv\_nxt && tp->ucopy\_len &&
3158           sock\_owned\_by\_user(sk) && !tp->urg\_data) {
3159          int chunk = min\_t(unsigned int, skb->len,
3160                      tp->ucopy\_len);
3161
3162          __set\_current\_state(TASK\_RUNNING);
3163          local\_bh\_enable();
3164          if (!skb\_copy\_datagram\_iovvec(skb, 0,
3165                      tp->ucopy\_iov, chunk)) {
3166             tp->ucopy\_len -= chunk;
3167             tp->copied\_seq += chunk;
3168             eaten = (chunk == skb->len && !th->fin);
3169             tcp\_rcv\_space\_adjust(sk);
3170          }
3171          local\_bh\_disable();
3172     }
```

Exercise: If \texttt{tp->ucopy\_task == current}, how can it \textit{not} be running?? Here the copy to user space doesn’t necessarily have to copy the entire segment, but it might. Even if it does \textit{eaten} can’t be set if the segment also contains a \textit{fin}.
Adding the buffer to the receive queue.

The value of "eaten" is set to 1 if all of the data in this segment was copied to user space. Otherwise eaten will be -1 (no data copied) or 0 some but not all data copied.

If eaten is <= 0 then the packet will be queued on the receive queue. Queue pruning details are not clear. In the Linux implementation of TCP buffer quota's appear to be somewhat soft and subject to change as a function of global memory demand.

```c
3174   if (eaten <= 0) {
3175     queue_and_out:
3176       if (eaten < 0 &&
3177          (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf
3178           || !sk_stream_rmem_schedule(sk, skb))) {
3179         if (tcp_prune_queue(sk) < 0 ||
3180            !sk_stream_rmem_schedule(skb, skb))
3181           goto drop;
3182       }
3183       sk_stream_set_owner_r(skb, sk);
3184       __skb_queue_tail(&sk->sk_receive_queue, skb);
3185     }
3186     tp->rcv_nxt = TCP_SKB_CB(skb)->end_seq;
```

As noted earlier TCP uses a variable ACK delay strategy that is implemented in tcp_event_data_recv().

```c
3187   if(skb->len)
3188     tcp_event_data_recv(sk, tp, skb);
3189   if(th->fin)
3190     tcp_fin(skb, sk, th);
```
Checking for filled holes

If the out of order queue is not empty this segment may have filled a hole that necessarily exists between the end of the receive queue and the start of the out of order queue. It may now be possible to move more segments from the out of order queue to the receive queue. The tcp_ofo_queue() function handles this. It is also possible for holes to be filled internal to the O-o-O queue, but these events are irrelevant.

```c
if (!skb_queue_empty(&tp->out_of_order_queue)) {
    tcp_ofo_queue(sk);
    /* RFC2581. 4.2. SHOULD send immediate ACK, when
     * gap in queue is filled.
     */
    if (skb_queue_empty(&tp->out_of_order_queue))
        inet_csk(sk)->icsk_ack.pingpong = 0;
}
if (tp->rx_opt.num_sacks)
    tcp_sack_remove(tp);
tcp_fast_path_check(sk, tp);
if (eaten > 0)
    __kfree_skb(skb);
else if (!sock_flag(sk, SOCK_DEAD))
    sk->sk_data_ready(sk, 0);
return;
```
In this case there is no data in the segment that can be used. It's necessary to ack but the segment must be dropped.

```c
3214    if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
3215       /* A retransmit, 2nd most common case. Force an immediate ack. */
3216       NET_INC_STATS_BH(LINUX_MIB_DELAYEACKLOST);
3217       tcp_dsack_set(tp, TCP_SKB_CB(skb)->seq,
                       TCP_SKB_CB(skb)->end_seq);
3218
3219     out_of_window:
3220     tcp_enter_quickack_mode(sk);
3221     inet_csk_schedule_ack(sk);
3222    drop:
3223       __kfree_skb(skb);
3224       return;
3225    }
```
received segment outside window

3227    /* Out of window. F.e. zero window probe. */
3228    if (!before(TCP_SKB_CB(skb)->seq,
            tp->rcv_nxt + tcp_receive_window(tp)))
3229       goto out_of_window;
3230

received packet overlaps rcv_nxt

3231    tcp_enter_quickack_mode(sk);
3232
3233    if (before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt)) {
3234       /* Partial packet, seq < rcv_next < end_seq */
3235      SOCK_DEBUG(sk, "partial packet:
            rcv_next %X seq %X - %X\n",
              tp->rcv_nxt, TCP_SKB_CB(skb)->seq,
              TCP_SKB_CB(skb)->end_seq);
3236      tcp_dsack_set(tp, TCP_SKB_CB(skb)->seq, tp->rcv_nxt);
3237      tcp_receive_window(tp)
3238         goto out_of_window;
3239    }
3240
3241    /* If window is closed, drop tail of packet. But after
3242       * remembering D-SACK for its head made in previous line.
3243       */
3244    if (!tcp_receive_window(tp))
3245       goto out_of_window;
3246    goto queue_and_out;
3247  }
3248
3249    TCP_ECN_check_ce(tp, skb);
The packet may also be dropped due to full buffer quota.

```c
3251    if (atomic_read(&sk->sk_rmem_alloc) > sk->sk_rcvbuf ||
3252        !sk_stream_rmem_schedule(sk, skb)) {
3253       if (tcp_prune_queue(sk) < 0 ||
3254          !sk_stream_rmem_schedule(sk, skb))
3255          goto drop;
3256    }
3257
3258    /* Disable header prediction. */
3259    tp->pred_flags = 0;
3260    inet_csk_schedule_ack(sk);
3261    SOCK_DEBUG(sk, "out of order segment: rcv_next %X seq %X -
3262     %X\n", tp->rcv_nxt, TCP_SKB_CB(skb)->seq,
3263     TCP_SKB_CB(skb)->end_seq);
3264    sk_stream_set_owner_r(skb, sk);
```
Adding to the out-of-order queue

Handling an out of order segment occurs here. If the out of order queue is presently empty then this packet is added and SACK data is constructed.

```c
3267 if (!skb_peek(&tp->out_of_order_queue)) {
3268     /* Initial out of order segment, build 1 SACK. */
3269     if (tp->rx_opt.sack_ok) {
3270         tp->rx_opt.num_sacks = 1;
3271         tp->rx_opt.dsack = 0;
3272         tp->rx_opt.eff_sacks = 1;
3273         tp->selective_acks[0].start_seq =
3274             TCP_SKB_CB(skb)->seq;
3275         tp->selective_acks[0].end_seq =
3276             TCP_SKB_CB(skb)->end_seq;
3277     } else {
3278         struct sk_buff *skb1 = tp->out_of_order_queue.prev;
3279         u32 seq = TCP_SKB_CB(skb)->seq;
3280         u32 end_seq = TCP_SKB_CB(skb)->end_seq;
3281     }__skb_queue_head(&tp->out_of_order_queue,skb);
3282     goto add_sack;
3283     /* Common case: data arrive in order after hole. */
3284     tp->selective_acks[0].end_seq = end_seq;
3285     return;
3286     goto add_sack;
3287 }
3288 else {
3289     struct sk_buff *skb1 = tp->out_of_order_queue.prev;
3290     u32 seq = TCP_SKB_CB(skb)->seq;
3291     u32 end_seq = TCP_SKB_CB(skb)->end_seq;
```

If the out of order queue is not empty then its necessary to figure out where to put this segment. The most likely place is on the end of the queue.

```c
3283      if (seq == TCP_SKB_CB(skb1)->end_seq) {
3284          __skb_append(skb1, skb, &tp->out_of_order_queue);
3285      if (!tp->rx_opt.num_sacks ||
3286          tp->selective_acks[0].end_seq != seq)
3287          goto add_sack;
3288      goto add_sack;
3289      /* Common case: data arrive in order after hole. */
3290      tp->selective_acks[0].end_seq = end_seq;
3291      return;
3292  }
```
Not last segment on out of order queue

If this is not the last segment then a backward search through the queue must be done to find where to put it.

3295       /* Find place to insert this segment. */
3296       do {
3297          if (!after(TCP_SKB_CB(skb1)->seq, seq))
3298             break;
3299       } while ((skb1 = skb1->prev) !=
3300           (struct sk_buff*)&tp->out_of_order_queue);

It's possible that overlap of entire segments may occur. This may result in dropping this segment if all data it carries is already on the O-o-O queue. What happens if all of the data in this segment is presently contained in TWO segments in the O-o-O queue???

3301
3302       /* Do skb overlap to previous one? */
3303       if (skb1 != (struct sk_buff*)&tp->out_of_order_queue &&
3304           before(seq, TCP_SKB_CB(skb1)->end_seq)) {
3305          if (!after(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
3306             /* All the bits are present. Drop. */
3307             __kfree_skb(skb);
3308             tcp_dsack_set(tp, seq, end_seq);
3309             goto add_sack;
3310          }
3311       }
3312       else {
3313           tcp_dsack_set(tp, seq, TCP_SKB_CB(skb1)->end_seq);
3314       } else {
3315           skb1 = skb1->prev;
3316       }
3317     }
3318     __skb_insert(skb, skb1, skb1->next,
3319         &tp->out_of_order_queue);
Or it is also possible that this segment covers multiple existing ones that may now be discarded.

```c
3320       /* And clean segments covered by new one as whole. */
3321       while ((skb1 = skb->next) !=
3322              (struct sk_buff*)&tp->out_of_order_queue &&
3323              after(end_seq, TCP_SKB_CB(skb1)->seq)) {
3324              if (before(end_seq, TCP_SKB_CB(skb1)->end_seq)) {
3325                 tcp_dsack_extend(tp,
3326                        TCP_SKB_CB(skb1)->seq, end_seq);
3327                 break;
3328              }
3329              __skb_unlink(skb1, &tp->out_of_order_queue);
3330              tcp_dsack_extend(tp, TCP_SKB_CB(skb1)->seq,
3331                        TCP_SKB_CB(skb1)->end_seq);
3332          __kfree_skb(skb1);
3333      }  
3334    add_sack:
3335        if (tp->rx_opt.sack_ok)
3336            tcp_sack_new_ofo_skb(sk, seq, end_seq);
3337      }
3338```